Efficient Out-of-core and Out-of-place Rectangular Matrix Transposition and Rotation

Paul Godard, Vincent Loechner, Cédric Bastoul

Abstract—Modern computers keep following the traditional model of addressing memory linearly for their main memory and out-of-core storage. While this model allows efficient row access to row-major 2D matrices, it introduces complexity to perform efficient column access. A common strategy to improve these accesses is to transpose or rotate the matrix beforehand, thus the accessing complexity is centralized in one transformation operation. Further column accesses are performed as row accesses to the transposed matrix therefore they are optimized to the memory model. In this paper, we propose an efficient solution to perform in-memory or out-of-core rectangular matrix transposition and rotation by using an out-of-place strategy, reading a matrix from an input file and writing the transformed matrix to another (output) file. An originality of our processing algorithm is to rely on an optimized use of the page cache mechanism. It is parallel, optimized by several levels of tiling and independent of any disk block size. We evaluate our approach on five common storage configurations: HDD, hybrid HDD-SSD, SSD, software RAID 0 of several SSDs and NVMe. We show that it brings significant performance improvement over a hand-tuned optimized reference implementation developed by the Caldera company and we confront it against the baseline speed of a straight file copy.

Index Terms—out-of-core, out-of-place, matrix transposition, matrix rotation, SSD, NVMe

1 Introduction

Efficient out-of-core matrix transposition and rotation is of utmost importance in many applications, e.g., FFT, K-Means clustering or image processing. The volume of data handled by precise simulations and high resolution pictures is in constant growth with the increase of computing power and storage size, both in main memory and secondary storage. The well-known class of *out-of-core* problems manipulates data that do not fit in memory.

The matrix transposition problem has been extensively tackled by previous research [1], [2], [3], [4], [5], [6], [7]. The traditional model of addressing memory is linear and benefits from spatial locality (in data blocks, to exploit hard drive and file system mechanisms or at a smaller granularity in caches), therefore row accesses benefit from row-major storage and column accesses benefit from column-major storage. For a given algorithm, it is essential for performance to decide how to store a matrix: in row-major or in column-major format. However, if an algorithm or a whole program requires both row accesses and column accesses to a given matrix, it may be useful to store the matrix in both formats. The transposition converts a matrix from one format to the other and centralizes the problem of inefficient accesses into this single part of the program.

The matrix 90 degrees clockwise rotation problem is related to the transposition problem: a rotation is a transposition combined with a horizontal reflection (see Fig. 1); hence the only difference is the reverse order of the elements in the destination rows (in row-major format). Such rotation algorithms are common in image manipulation problems, for example to convert a landscape image to portrait or the other way around for appropriate processing or printing. In the following of this paper, a *matrix transformation* refers to either matrix transposition or rotation.

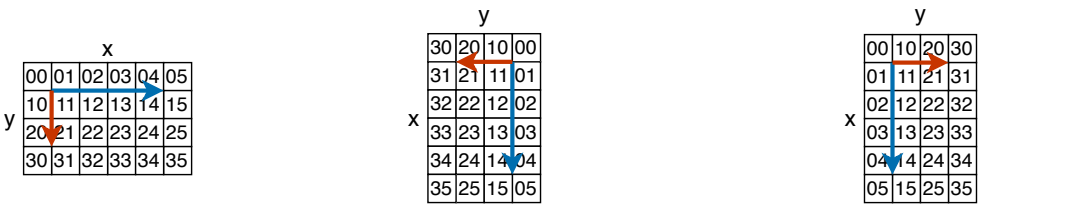
Our primary target is efficient matrix transformation in the context of deep image processing pipelines for high-definition professional printing. After rasterization, we manipulate rectangular bitmap images with a typical data size of dozens of GB. As the raster image processor runs on off-the-shelf hardware, our data is typically too large to fit in main memory but easily fits twice in secondary storage. This situation may hold in other contexts where a user manipulates large datasets organized as matrices.

We address the problem of *out-of-place* matrix transformation where the input matrix is given as a file on secondary storage and the user wants to generate the transformed output matrix as a new file on secondary storage. It opposes to *in-place* transformation, where the original file is modified incrementally up to turn into the output matrix.

The secondary storage technology significantly evolved recently with the popularization of flash memory drives (SATA/SSD and NVMe). Compared to a traditional hard disk drive (HDD) requiring the actuator arm to move and the disk to spin, a flash memory has no moving component. It is much faster and permits efficient random accesses, although sequential writes are more efficient than random writes to some extent.

We propose a new algorithm to perform out-of-core and out-of-place rectangular matrix transformation that is very efficient on SSD, SSD based RAID 0 and NVMe. At coarse-grain, it is a tile-by-tile source file reading and destination file writing that benefits from the operating system efficient page cache implementation, enforced by some standard POSIX system calls. The fine grain computation is done by prioritizing contiguous data writes, in parallel and strip-mined for efficient cache usage and automatic vectorization. We present experimental evidence that our algorithm

- *Paul Godard is with Caldera, the University of Strasbourg and Inria*
- *Vincent Loechner and Cédric Bastoul are with the University of Strasbourg and Inria*



(a1) Original Layout (b1) Rotated Layout (c1) Transposed Layout

y: row index
x: column index

(a2) Original Position

```
for (x = 0; x < W; x++)
  for (y = 0; y < H; y++)
    dst[x][y] = src[H-1-y][x];
```

(b2) Naive Rotation

```
for (x = 0; x < W; x++)
  for (y = 0; y < H; y++)
    dst[x][y] = src[y][x];
```

(c2) Naive Transposition

$$i = yW + x$$

(a3) Original Index

$$i' = (iH + H) \bmod (HW + 1) - 1$$

(b3) Rotation Mapping

$$i' = \begin{cases} HW - 1, \text{if } i = HW - 1 \\ iH \bmod (HW - 1), \text{otherwise} \end{cases}$$

(c3) Transposition Mapping

Fig. 1. Matrix Rotation and Transposition: data layout transformation examples, naive codes and linear mappings for $H \times W$ matrices

reaches near the speed of a simple file copy.

The general problem is presented in Sec. 2. We give some related work in Sec. 3. In Sec. 4 we present our solution, and we describe the benchmarks in Sec. 5. Finally, we conclude and discuss future work in Sec. 6.

2 Problem Statement

This section presents the background of our work. It details our notations and general assumptions, then it reviews the matrix transposition/rotation problems and provides some characteristics of the memory systems we are addressing.

In this work we consider a $H \times W$ source matrix `src`, where each element is specified using *zero-based* indexing, i.e., the first element is `src[0][0]` and the last element is `src[H-1][W-1]`. Elements are stored contiguously according to a *row-major* organization: rows are stored one after the other. We address general rectangular matrices without specific size constraints or padding.

Given a $H \times W$ matrix `src`, a rotation is a data layout transformation so that the matrix is rotated by 90 degrees: elements of Fig. 1(a1) are rearranged to Fig. 1(b1). A transposition is a related data layout transformation so that each row (resp. column) is transformed to a column (resp. row): elements of Fig. 1(a1) are rearranged to Fig. 1(c1). Rotated and transposed matrices differ only by a reversal of their rows. Naive kernels to perform these operations are shown in Fig. 1(b2) and Fig. 1(c2). They implement a mapping transformation associating to each index of the original layout a new index in the output layout. The mapping formulas are presented in Fig. 1(b3) and Fig. 1(c3).

When the matrices are too large to fit in the main memory, they come as files stored on a secondary storage, either a traditional spinning hard drive (HDD), a flash memory drive (SSD, NVMe) or even several of them striped in a RAID 0 logical unit. On the one hand, random accesses on HDDs are much slower than sequential accesses because moving the mechanical arm is extremely slow. On the other hand, flash memory drives provide an efficient support for random accesses with an asymmetry between read and write accesses, write accesses being slower. Our context makes it significant to provide efficient solutions in all these cases despite their different properties. In particular, since both device families are block-based, an efficient management of blocks is of utmost importance.

Most modern operating systems provide an abstraction layer to access block devices data [8]. This abstraction offers both portability and automatic optimization to access these slow devices. In the Linux kernel, by default, all block devices are accessed through the main-memory *page cache*. The page cache system improves data reuse, speeds up writes by executing these operations on cached data and speeds up further reads with speculative read ahead.

While the page cache improves the performance of a wide range of applications, they are harmful to the naive out-of-core matrix transformations. They drastically increase the amount of data transferred from and to the disks since data locations, contained in many different pages, are accessed successively thus causing many page faults and synchronizations. The resulting poor performance is visible in our experiments in Sec. 5.

3 Related work

For in-memory problems, naive implementations of matrix rotation/transposition lead to poor data locality and non-optimized accesses. For out-of-core problems, they lead to a large amount of costly I/O operations. Due to its importance, the matrix transposition optimization problem has been widely addressed in previous work.

We may distinguish two classes of techniques to address out-of-core transposition. The first one is *block matrix transposition*. Its basic idea is to consider the matrix as a block matrix where each block can fit in memory, transpose the individual blocks in memory then transpose the block matrix [2], [9]. The second one is *multistage matrix transposition*, first introduced by Eklundh [1] for in-place transposition: at each stage a given number of rows are read to memory where appropriate elements are rearranged, then the rows are written back to external memory. After a number of stages, the matrix is fully transposed with an optimized number of disk accesses. Kaushik et al. [2] improved Eklundh's approach by combining reads to reduce the number

of I/O operations. Suh and Prasanna [5] further improved out-of-place transposition by scheduling and balancing I/O operations and by collecting data to buffers with simple access patterns to reduce the index computation time. Krishnamoorthy et al. [3], [6] exploited I/O characterization and parallelism in the context of distributed matrices. Our work pertains to the block matrix class of solutions. Differently from existing approaches, we exploit the memory paging provided by the operating system to organize the writing operations, and thus benefit from the low-level operating system disk accesses optimizations.

Optimizing the in-memory part of out-of-core transposition may lead to second order yet non-negligible performance improvement. Gustavson et al. [7] presented an efficient parallel approach for in-place transposition of rectangular matrices using a greedy determination of cycle leaders to achieve load balancing and compete with out-of-place conversion. Zekri [4] exploited vector instructions to optimize the transposition very internal part. Our technique exploits parallelism by achieving a parallel block processing and parallel writes to virtual memory that completely hides the in-memory processing time. The compiler automatically generates vector instructions when the appropriate optimization flags are set. So both these concerns are included and improved by our method.

Most existing out-of-core transposition strategies suppose that matrices are stored on HDD with their specific performance issues, such as poor random access performance. Shao et al. [10] avoid rotational latency and minimize the access time of neighboring dataset blocks by using a data placement strategy providing efficient semi-sequential accesses along the outer dimensions of a multidimensional array. Thonangi and Yang [11] exploit SSD characteristics such as efficient random access and asymmetry between read and write operation performance to address general data permutations. Since it manages I/O operations explicitly, this solution depends on the SSD characteristics to be efficient, while we exploit the operating system for a better performance portability. Moreover, the general nature of their technique, based on key-value pairs, uses extra data structures and makes it less efficient at solving simpler specialized processing such as matrix transposition or rotation.

4 Efficient Transposition and Rotation

In this section, we detail which optimization we apply to the naive code in Fig.1(b2): how to choose the iteration execution order, where to tile and to parallelize, how and when to read the input file and to write the output file.

4.1 Preliminary ideas

The first observation is that the naive loop nest does not carry any dependence so we are entirely free to reorder it by applying any transformation. In our case we will be using tiling, interchange, strip mining and parallel loops.

The first idea of avoiding reading the entire input file in memory and writing it back to the disk after transformation is obvious: if the file is larger than the available memory, this will induce paging and page faults that drastically limit performance. The input will be read as successive parts using a temporary fixed size buffer. As a consequence, the original loop nest should be tiled and each tile execution will first read a part of the input file, perform the transformation and then write the corresponding part of the output file.

The second idea is that ordered writes are much more efficient than out-of-order writes. We observed in a few experiments that writing in order to the output file matters much more than reading in order from the input file for the best performance on SSD drives, which corresponds to the expected behavior: random access reads are efficient on SSD drives, contrary to random writes. On HDD drives ordered writes should be slightly favored over ordered reads but the difference in performance is smaller than on SSD drives. For the best performance, the algorithm should write the `dst` array to the output file roughly in order.

Finally, in another preliminary experiment we observed that writing an output file through its memory mapping (achieved using POSIX's `mmap` function) has the same performance as explicit writes from an intermediate buffer. As detailed at the end of this section, some standard POSIX parameters have to be set for this to perform best. This policy allows us to avoid taking care about writing the file explicitly: the output file simply has to be `mmap`'ed to the destination buffer and the operating system will manage it efficiently and using a minimal amount of memory. By writing each data to the destination buffer only once and in sequential order, we maximize the probability that the oldest pages are complete when the page replacement algorithm will write them to the output file and unmap them from memory. This strategy permits the operating system to schedule the disk accesses in an efficient way. Another benefit of `mmap` is that we spare an extra buffer for a temporary destination array, or the need to perform complex memory copies in an in-place memory transformation.

The input file is explicitly read into a buffer and not `mmap`'ed like the output file, for two reasons: (a) to avoid the many and costly page faults interruptions due to the column-major access pattern of the input file; and (b) to separate (explicit) reads from (implicit) writes. This separation allows our algorithm to alternate reading and writing, which simplifies operating system's I/O scheduling.

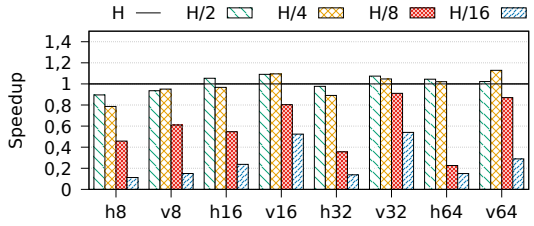
4.2 Main algorithm

In order to control the maximum size of a memory buffer, the original `x,y` loop nest of Fig.1(b2) has to be tiled to divide the `dst` array in fixed-sized rectangular tiles. The tiles are then scanned in the same order (`x`-outer loop).

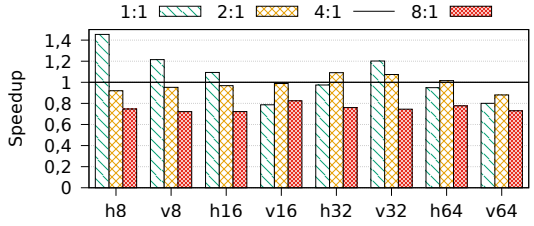
The tile size along the `y` dimension should be larger than the tile size along `x` in order to favor contiguous writes. To compute the tile sizes, we first divide the amount of memory taken by the array by the limit of available memory to get the minimal number of necessary tiles. Then the tile size in each dimension is determined by favoring a large `y_tile` size over a smaller `x_tile` size. We conducted experiments on a 8 GB main memory platform and various matrix configurations reported in Tab. 1 (see Sec. 5 for details). On SSDs, these experiments suggest to select the maximal possible `y_tile` size, i.e. the matrix height `H`, as shown in Fig. 2a. This figure shows that the baseline `y_tile=H` is on average better than the other values. This is the best choice

TABLE 1
Matrix Characteristics

Name	Abbreviation	Width	Height	Ratio (w/h)
horizontal 8 GB	h8	125k	64k	1.95
vertical 8 GB	v8	64k	125k	0.51
horizontal 16 GB	h16	200k	80k	2.50
vertical 16 GB	v16	80k	200k	0.40
horizontal 32 GB	h32	256k	125k	2.05
vertical 32 GB	v32	125k	256k	0.49
horizontal 64 GB	h64	400k	160k	2.50
vertical 64 GB	v64	160k	400k	0.40



(a) SSD: `y_tile` value, with `H` the matrix height



(b) HDD: `y_tile:x_tile` ratio

Fig. 2. Average speedup for different tile sizes with respect to our selected choice as baseline, with a 1 GB `src` buffer on different matrix shapes (horizontal and vertical) and sizes (from 8 to 64 GB, see Tab. 1)

because it favors ordered writes over ordered reads. On HDDs a `y_tile`:`x_tile` ratio of 4:1 is the best compromise we have found in our experiments, as shown in Fig. 2b. The baseline 4:1 is on average better than the other ratios. In this configuration ordered reads also affect performance, but to a lower extent (about 4 times less than ordered writes).

Each tile execution starts by reading the necessary input data: a (`y_tile` × `x_tile`) rectangle is loaded from the input file into an allocated `src` buffer:

```
/* (x_tile, y_tile) = size of the current tile  */
/* (bx, by)         = current tile coordinates  */
/* assuming the arrays contain bytes            */
for(size_t y=0 ; y<y_tile ; y++) {
  /* input line position (by+y) in reverse tile
     order   */
  /* for the transposition use: (by+y) * W + bx */
  off_t offset = (H-1-(by+y_tile-1-y)) * W + bx;
  lseek(src_file, offset, SEEK_SET);
  read(src_file, &src[y][0], x_tile);
}
```

Then we transform the intra-tile loops to take advantage of parallelism, cache locality and SIMDization. The global order of a single `y` loop is preserved while tiling the `x`-loop around the `y`-loop as a parallel outer loop and an inner sequential loop. We get the `parallel_outer_x-y-inner_x` order:

```
/* (x_tile, y_tile) = size of the current tile */
```

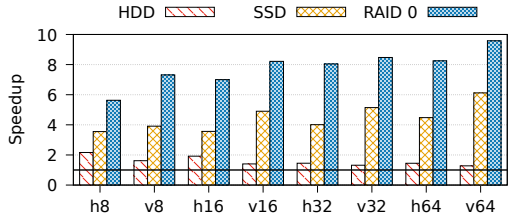


Fig. 3. Speedup due to the use of `posix_fadvise` with a 1 GB `src` buffer, on different matrix shapes and sizes (from 8 to 64 GB, see Tab. 1)

```
/* X loop */
#pragma omp parallel
#pragma omp for schedule(static, PAR_BLOCK)
for(size_t outx=0; outx<x_tile; outx+=INNER_X){
  const size_t inmaxx=MIN(outx+INNER_X, x_tile);
  /* Y loop */
  for(size_t y=0 ; y<y_tile ; y++ )
    /* X inner loop */
    for(size_t inx=outx ; inx<inmaxx ; inx++ )
      /* (bx, by) = current tile coordinates    */
      /* for the transposition use: src[y][inx] */
      dst[bx+inx][by+y] = src[y_tile-1-y][inx];
}
```

The advantages of this loop transformation are the following. The accesses to the `src` array take advantage of spatial locality and are vectorized by the compiler. The `dst` array is accessed `INNER_X` times (in different locations) by the `outx` and `y` loops. When `y` changes, it benefits from spatial locality. In our experiments an `INNER_X` tile size accessing 64 bytes of array data has been determined to perform best probably because it accesses one L1 cache line on our experimental platform. The `PAR_BLOCK` parameter controls the parallel granularity of the outer loop: each thread writes `PAR_BLOCK` × `INNER_X` successive rows in the file. We experimentally determined that `PAR_BLOCK=4` performs best on our platform, but its influence is very low. `PAR_BLOCK` and `INNER_X` would require adjustment on significantly different platforms since their impact depend on both the hardware and the operating system.

4.3 Fine-tuning

We evaluated an explicit synchronization at the end of each tile execution with an explicit call to `fsync()` to avoid asynchronous writes conflicting with the next tile reads. However we found out it was not necessary since the operating system is smart enough not to write the dirty pages to disk during the following reads.

In our final implementation, we use some POSIX system calls to ensure the best performance of file accesses. A fine grain advice is given using `posix_fadvise(..., POSIX_FADV_WILLNEED)` to limit the read aheads to the parts of the input file lines contained in the current tile. Multiple advises are called before the effective `read`'s in order to maximize the benefit of read ahead mechanism. This is done by groups of 64 successive advices followed by 64 reads that is handled efficiently by the Linux operating system in our experiments.

The speedup due to the use of these `posix_fadvise` calls is shown at Fig. 3 in some representative cases presented in detail in Sec. 5: the horizontal axis represents

different matrix sizes from 8 to 64 GB and for each storage configuration (HDD, SSD, RAID). As expected gains on HDD are not very important and are decreasing when the matrix size increases since the overhead of the HDD moving arm is the main limitation in this case. On SSD and RAID, we reach impressive speedups of more than 4x and 7x (respectively) on average, thanks to the low latency of random accesses on these devices.

The last two optimizations that we made are the following. The output file is directly mapped to memory on the `dst` array using a `mmap()` call. Then the system is said to perform aggressive sequential pages read ahead and writes using `posix_madvise(..., POSIX_MADV_SEQUENTIAL)`. In this way, we ensure that the memory is not polluted by pages of the `dst` array that will not be accessed again. Finally, the `src` buffer is said to be kept in main memory by a call to `posix_madvise(..., POSIX_MADV_WILLNEED)`.

4.4 Hybrid HDD-SSD transformation

When transforming a matrix from an input file stored on HDD to an output file stored on the same HDD, it can be useful to use a secondary SSD drive as a bridge. This basic idea indeed outperforms the direct transformation on the same drive, despite its overhead due to an extra file copy. The gain comes from (a) reading a file on one drive while writing on another avoids any conflict between reads and writes, and (b) the transformation itself is much more efficient when using an SSD.

There are two alternative ways of realizing this hybrid transformation:

1) perform the transformation when transferring the file from HDD to SSD and then copy the resulting file back to SSD: HDD $\xrightarrow{T}$ SSD $\rightarrow$ HDD;
2) copy the original file from HDD to SSD without any modification and then perform the transformation during the transfer back from SSD to HDD: HDD $\rightarrow$ SSD $\xrightarrow{T}$ HDD.

As expected, the most efficient solution is the second one. During the first straight copy, both files are accessed sequentially so we perform ordered reads on HDD on one side and ordered writes on SSD on the other. During the transformation phase, we perform random read accesses to the SSD and mostly ordered writes to HDD. Since random accesses on SSD are more efficient than on HDD and since random *read* accesses are more efficient than random write accesses on SSD, this is obviously the best solution.

Nevertheless we experimented both solutions and measured their speedup compared to the baseline HDD to HDD transformation time. The results are presented in Fig. 4, in the same condition as in the previous figure: the horizontal axis corresponds to different matrix sizes and shapes and the buffer size is 1 GB. The average speedup of solution (1) is 1.8$\times$ and for solution (2) it is 4.3$\times$. The second solution is the one that we implemented and which results are shown and discussed in the next section (this HDD-SSD configuration is named *"hybrid configuration"*).

The final user can expect an acceleration of up to 8.7$\times$ when using a bridge SSD drive for performing the out-of-place transformation of a matrix stored on HDD.

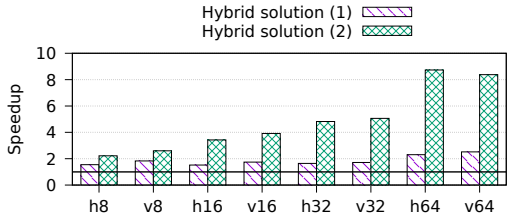


Fig. 4. Speedup due to the use of hybrid HDD-SSD configuration against single HDD with a 1 GB `src` buffer, on different matrix shapes and sizes (from 8 to 64 GB, see Tab. 1)

TABLE 2
Maximal read and write transfer rates (MB/s)

	HDD	SSD	RAID 0 (4 SSD)	NVMe
Reading speed	135	535	1700	1500
Writing speed	125	310	1200	1200

5 Benchmarks

5.1 Hardware Setup

We evaluated the benchmarks on five platforms with different storage configurations. Platform *HDD* has a single HDD, *hybrid* an HDD and an SSD, *SSD* has a single SSD, *RAID 0* a software RAID 0 of four identical SSDs managed by the `mdadm`[1] tool, and *NVMe* has a single NVMe SSD. The filesystem is `ext4` in all configurations. The disks are:

- a 1 TB HDD at 7200RPM "HGST Travelstar 0J22423";
- four 256 GB SATA/SSDs "Transcend SSD370S";
- a 1 TB NVMe "PC601 SK hynix M.2".

Table 2 presents the maximal read and write transfer rates that we measured for each drive type, by sequentially accessing large chunks of data using the `dd` command.

The four first platforms are equipped with an Intel Xeon D-1521 and 8 GB of DDR4 RAM at 2133 MT/s. The NVMe platform has an Intel Core i9-9900 and 16 GB of DDR4 RAM at 2666 MT/s. They run Ubuntu 18.04.1 LTS with Linux kernel 4.15.0-43-generic. Gcc version 7.3.0 with options `-O3 -fopenmp` is used to compile the programs.

5.2 Protocol

Several runtime optimizations are provided by default by the kernel in order to improve the performance of the disk accesses [8] primarily based on making a cache system by exploiting unused RAM.

This is why duration or speed measures of disk-access bounded programs are highly affected by the cache used by the kernel for recently accessed files. Hence, before each of our run we explicitly clear this cache through the kernel interface[2] and instruct to write any data buffered in memory out to disk with an explicit `sync` command; this `sync` operation is also performed at the end of each benchmark and just before collecting the execution time. In this way, the benchmarks run from a clean and reproducible state.

1. Linux software raid.
2. `echo 3 >/proc/sys/vm/drop_caches`

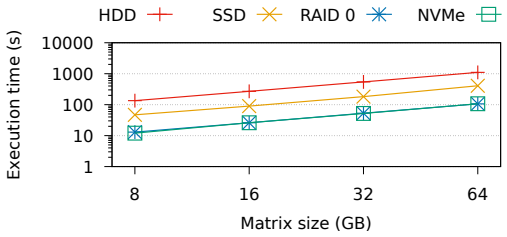


Fig. 5. Execution time of `cp` on different drive types

5.3 Measurements

Table 1 presents the characteristics of the different benchmarks. The matrix sizes are selected from the main memory size (8 GB) up to 8 times its size (64 GB). Since the matrices are rectangular, they are tested in both orientations. Their width over height ratio is chosen to create a rectangle around twice wider (horizontal) or narrower (vertical).

We compared three different implementations to perform the matrix rotation:

- `Simple` implements the naive algorithm presented in Fig. 1(b2), by performing the rotation element by element with two nested loops reading from and writing to `mmap`'ed files;
- `Caldera` is a hand-tuned application developed by the Caldera company to achieve a fast rotation of out-of-core pictures represented by a matrix per color channel. This professional implementation uses an out-of-core and out-of-place technique minimizing the amount of data read from the input file and performing an in-memory tiled rotation to maximize the sequential data writing. This reference implementation is in use in the *CalderaRIP V12.0* software[3] ;
- `MaRTOO` (Matrix Rotation and Transposition Out-of-core Out-of-place) is our solution exploiting the algorithm presented in Sect. 4.

We did not find any other easily available optimized implementation of rectangular out-of-core out-of-place matrix transformation.

Unlike the simple implementation, both `Caldera` and `MaRTOO` require a buffer to perform partial rotation in-memory before writing it to the output. We ran experiments with three buffer sizes: 35 MB, 1 GB and 5 GB. These sizes match a very low memory usage, a regular one and a greedy one on our experimental platform, allocating up to 5 GB of the available 8 GB and leaving only 3 GB to the operating system (including the page-cache).

We used the `cp` program as a baseline for the execution time. The `cp` command performs a strict duplication of a file which is perfectly comparable to our out-of-place approach without rotation processing. We used the standard version of `cp` without any option from the *GNU coreutils 8.28*[4] provided by default on various Linux distributions. The execution time of `cp` is collected with the same care of a clean and reproducible state as presented above. We checked that the execution time of `cp` is linear in the file size on each drive type as shown in Fig. 5. We also observe that `cp` provides a throughput corresponding to the transfer rates observed in Table 2 (for reading the matrix and writing it back) which reflects its quality as a reliable baseline.

In our experiments the matrix elements are one byte long which corresponds to one channel of one pixel of an image. In this way the matrix size in bytes is equal to the number of elements to process. Our preliminary experiments have shown that changing the data size while keeping the same quantity of processed data does not impact the execution times. The results presented in this section are consistent among various matrix data types or sizes.

We observed the amount of data read from and written to the drives by accessing the statistics available through the `/proc/<pid>/io/` interface of the Linux kernel.

5.4 Results

The execution times of our benchmarks in the five configurations are presented in Figures 6–10. Notice that in all these figures, the Y-axis is logarithmic, in order for the plots to be compact and easily readable but which flattens the gaps: a difference of one unit between two bars represents an acceleration of 10×. They compare the three implementations for different buffer sizes. The baseline is the `cp` execution time. The results of the simple implementation and the `cp` baseline are reproduced three times in each figure since they are not dependent on the buffer size.

We did the experiments both on the transposition and the rotation algorithms but the results show similar behavior so we only present the rotation here. The missing values for the simple implementation correspond to execution times exceeding 10 hours. It shows its limits to process out-of-core matrices due to the high pressure put on the kernel page cache, partially accessing many different device blocks. For example, we observed that it reads up to 64 kB to write one single byte for *v16* in Fig.6!

In the HDD configuration (Fig.6) `MaRTOO` completes all benchmarks with a near-linear execution time w.r.t. the matrix size. When the buffer is very small (35 MB) `MaRTOO` has similar performance than the one of `Caldera`. Due to the small buffer size, the non-contiguous disk accesses are unavoidable since we need to read and write the files in different orders. When the buffer size increases (1 GB and 5 GB) `MaRTOO` outperforms `Caldera` since it has a higher contiguous versus non-contiguous disk access ratio. On the matrices *h8* and *v8* with a buffer size of 5 GB, we observe that the transformation is almost as fast as the baseline straight copy. In this particular case, the matrices are scanned by only two tiles: the input file is scanned (by the HDD arm) only twice and the output file is written in order. For larger matrix sizes the number of tiles increases and consequently the number of times each file has to be scanned increases too, degrading the performance.

In comparison, Fig. 7 shows the results obtained with the hybrid configuration. Against the HDD results presented in Fig. 6, both `Caldera` and `MaRTOO` implementations benefit from using an SSD drive as a bridge. Thanks to more efficient disk accesses, we outperform `Caldera` for all matrix and buffer sizes. When using a 5 GB buffer, the execution time speedup is 3.8× on average and up to 7.9× on large

3. https://www.caldera.com
4. https://www.gnu.org/software/coreutils/

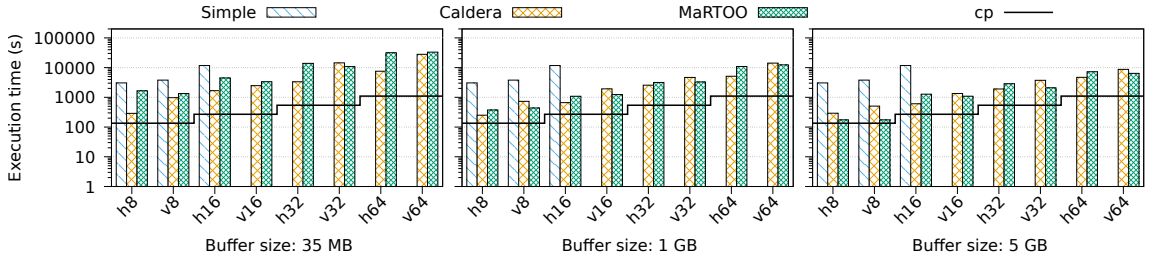


Fig. 6. Execution time on HDD configuration (`cp`: straight file copy time)

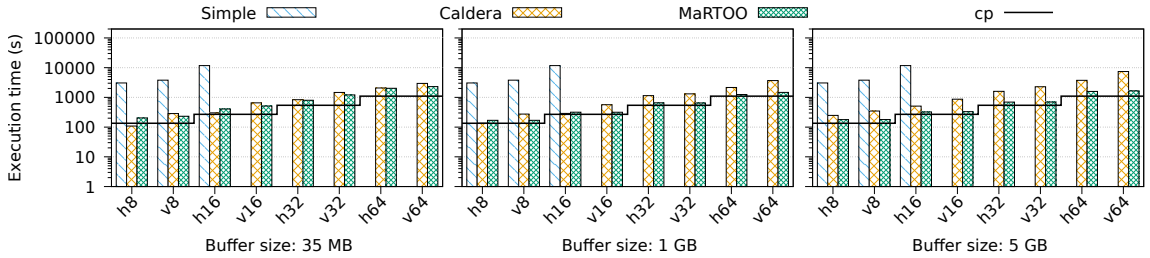


Fig. 7. Execution time on hybrid configuration (`cp`: straight file copy time)

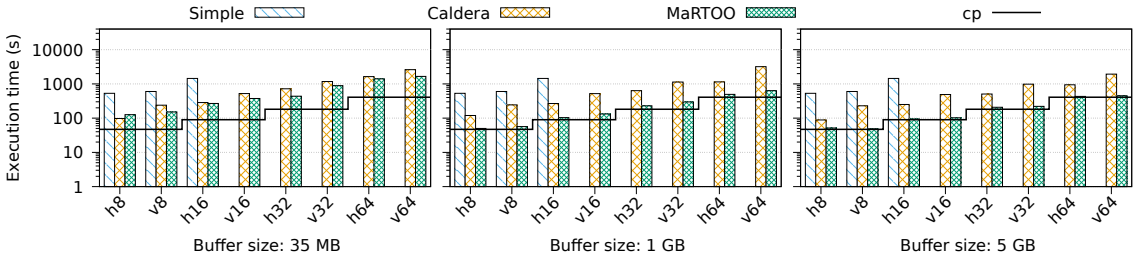


Fig. 8. Execution time on single SSD configuration (`cp`: straight file copy time)

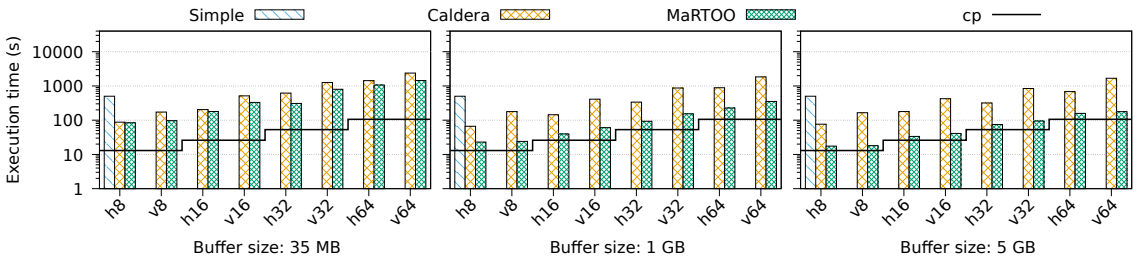


Fig. 9. Execution time on RAID 0 SSD configuration (`cp`: straight file copy time)

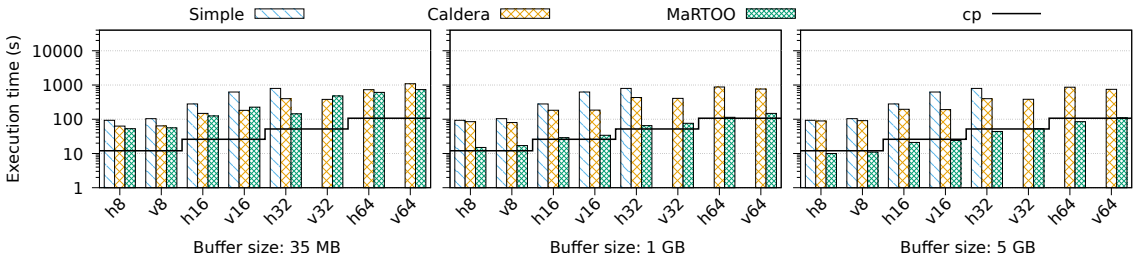


Fig. 10. Execution time on NVMe SSD configuration (`cp`: straight file copy time)

matrices. Remarkably, we obtain lower execution times with a 1 GB buffer. It appears that the Linux kernel performs more efficient disk accesses in this configuration since it has more page cache available (in the available main memory of 8 GB): it can flush dirty pages later and thus overlap the HDD output writes with the SSD input reads.

The results obtained with the SSD configuration presented in Fig. 8 show about 5× speedup on average for `MaRTOO` compared to the `Caldera` implementation for a 1 GB buffer. `MaRTOO` performs close to the optimal `cp` execution times the higher the buffer size: using a buffer of 5 GB the matrix rotation is only 10% slower on average than the straight file copy.

Figure 9 present the results on a RAID 0 of four SSDs. This plot illustrates that `Simple` performs even worse for this configuration. According to our analysis, this is due to the RAID 0 striping mechanism inducing more aggressive read ahead, which pollutes the page cache even faster. Similarly to the single SSD configuration, `MaRTOO` achieves near to the `cp` baseline performance and gets closer to this baseline the larger the buffer. We outperform `Caldera` by a factor of almost 10× on average.

Finally, Fig. 10 presents the results on the single NVMe platform. Those results resemble the ones of the SSD, since it is the same flash memory based technology, but four times faster thanks to the four times faster NVMe throughput. The difference with the `Caldera` implementation is even higher on this platform: on average `MaRTOO` outperforms it by 5.7× using a 1 GB buffer. The absolute best performance of NVMe is similar to the one of the RAID 0 configuration since they provide similar throughput. One can notice on this last graph, for a 5 GB buffer size, that our algorithm slightly outperforms the `cp` *baseline* in some cases. This is probably due to the fact that our algorithm uses multiple threads to write data to the file and thus achieves to saturate the NVMe bus, while `cp` is single-threaded and does not reach the maximum NVMe throughput.

We measured the effective amount of written and read data. The effective amount of data written to the drives by `MaRTOO` for the 64 GB matrices is close to the output file size with an average of 6% of writing surplus on the single HDD configuration and less than 1% on the single SSD and RAID 0 configurations. The writing surplus is due to the kernel synchronizing partial block writes. It is higher on single HDD due to the tiling pattern which generates more non-contiguous output rows. In comparison, with the same matrices, the `Caldera` implementation achieves an average of 16% of writing surplus on HDD configuration, and 10% on SSD and RAID 0.

Regarding the effective amount of reads, on the largest matrices (64 GB) `MaRTOO` generates an average of 52% (up to 77%) read surplus on HDD, an average of 20% (up to 32%) on single SSD and an average of 28% (up to 65%) on RAID 0. The read surplus is due to the unused data in blocks overlapping the border of the tiles, unused data read ahead by the kernel and the partial writes mentioned above which requires reading previously partially written blocks to complete. In comparison, with the same matrices, the `Caldera` implementation achieves an average of 53% (up to 126%) of read surplus on HDD configuration, 56% (up to 135%) on SSD, and 64% (up to 194%) on RAID 0.

6 Conclusion

This paper presents a new approach to tackle efficient out-of-core and out-of-place rectangular matrix transposition and rotation. It relies on a block-matrix strategy with a parallel, cache-efficient intra-tile processing and an original in-memory file mapping with an adequate tile scheduling to exploit efficiently the operating system page cache mechanism on a large variety of secondary storage technologies. Taking advantage of the efficient random read access offered by flash memory drives while respecting their propensity for efficient sequential write access, our technique is about 10 times faster than a reference implementation on a RAID 0 SSD configuration and more than 5 times faster on a single SSD and NVMe configurations. In many cases, its performance gets close to the baseline performance of a file copy.

Taking advantage of the operating system provided page cache mechanism and POSIX calls our proposal offers good relative independence to the file system or disk low-level parameters and good performance portability.

Ongoing work aims at improving performance by implementing a better usage of the blocks that are read by the kernel but not completely used by a tile, *i.e.*, reading rows dynamically sized to the disk-block boundaries from the input file and reuse these data.

Acknowledgments

This work was supported by the Caldera company.

References

[1] J. O. Eklundh, "A fast computer method for matrix transposing," *IEEE Transactions on Computers*, vol. 21, no. 7, pp. 801–803, 1972.

[2] S. D. Kaushik, C.-H. Huang, R. W. Johnson, P. Sadayappan, and J. R. Johnson, "Efficient transposition algorithms for large matrices," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. ACM, 1993, pp. 656–665.

[3] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C.-C. Lam, and P. Sadayappan, "Efficient parallel out-of-core matrix transposition," *International Journal on High Performance Computing and Networking*, vol. 2, no. 4, pp. 110–119, 01 2006.

[4] A. Zekri, "Restructuring and implementations of 2d matrix transpose algorithm using sse4 vector instructions," in *2015 International Conference on Applied Research in Computer Science and Engineering (ICAR)*, Oct 2015, pp. 1–7.

[5] J. Suh and V. K. Prasanna, "An efficient algorithm for out-of-core matrix transposition," *IEEE Transactions on Computers*, vol. 51, no. 4, pp. 420–438, April 2002.

[6] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan, "Efficient parallel out-of-core matrix transposition," in *2003 Proceedings IEEE International Conference on Cluster Computing*, Dec 2003, pp. 300–307.

[7] F. Gustavson, L. Karlsson, and B. Kågström, "Parallel and cache-efficient in-place matrix storage format conversion," *ACM Trans. Math. Softw.*, vol. 38, no. 3, pp. 17:1–17:32, Apr. 2012.

[8] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, 3rd Edition: from I/O ports to process management*. "O'Reilly Media, Inc.", 2005.

[9] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988.

[10] M. Shao, S. Schlosser, S. Papadomanolakis, J. Schindler, A. Ailamaki, and G. Ganger, "MultiMap: Preserving disk locality for multidimensional datasets," in *International Conference on Data Engineering*, April 2007, pp. 926–935.

[11] R. Thonangi and J. Yang, "Permuting data on random-access block storage," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 721–732, Jul. 2013.